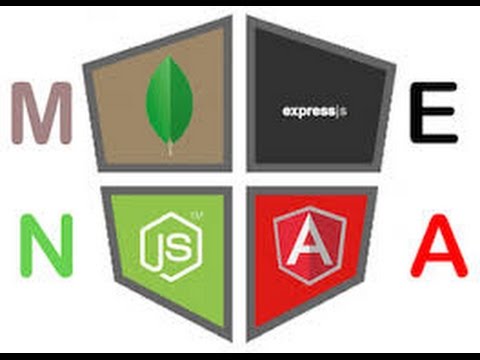
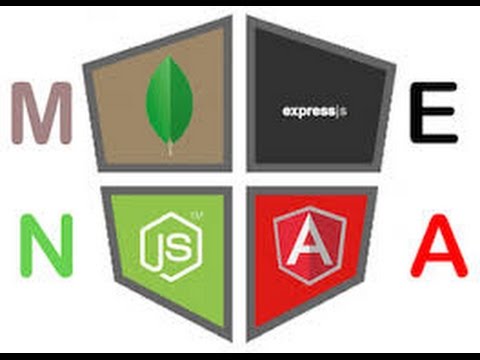
**Exercise06\_03\_01 – Step 1**

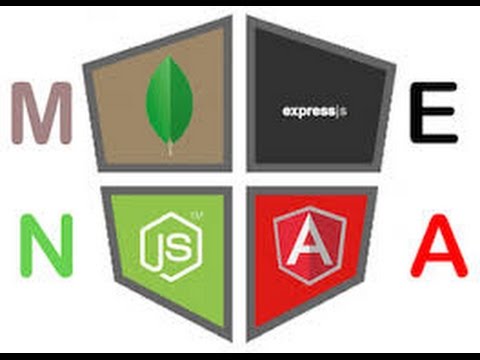
1. We will implement User Registration for our application so that users may be ***authenticated***. We will start by creating a new View for Registration. Let’s move over to the front-end ***src/app/components/navbar*** and open up ***navbar.html***.
2. Let’s modify the standard boilerplate to change the **<a>** hyperlink content to ***My Message Board***. Remove the entire ***<ul>*** and its contents that has the ***Application was created …*** text. Finally, change the ***<li>*** content from ***About*** to ***Login***, and ***remove*** the second ***<li>***. Our HTML should appear as follows:  
   ***<nav class="navbar navbar-static-top navbar-inverse">  
    <div class="container-fluid">  
    <div class="navbar-header">  
    <a class="navbar-brand"   
    href="https://github.com/Swiip/generator-gulp-angular">  
    <span class="glyphicon glyphicon-home"></span>   
    My Message Board  
    </a>  
    </div>  
     
    <div class="collapse navbar-collapse" id="bs-example-navbar-  
    collapse-6">  
    <ul class="nav navbar-nav">  
    <li class="active"><a ng-href="#">Home</a></li>  
    <li><a ng-href="#">Login</a></li>  
    </ul>  
    </div>  
    </div>  
   </nav>***Let’s run our ***server.js*** from the ***back-end***, then our ***front-end*** with ***gulp serve***. Turn on Developer Tools and the Console tab, and make sure we have no problems in either console.
3. We need to create our auth ***View*** and its ***Controller***, so let’s create a folder in ***src/app*** called ***auth***, with two files in it named ***auth.html*** and **auth.controller.js**. first we will add some ***placeholder*** text to ***auth.html*** as follows:  
   **hello**
4. We can begin to define our controller ***in auth.controller.js***. We will scaffold out the new controller ***class*** and ***export*** it:  
   ***export class AuthController {  
      
   }***
5. It will be necessary to ***register*** our Controller and View with the ***router***, so let’s open up ***index.route.js***. Copy the first ***.state*** down below itself, making sure to remove the ***semicolon*** between them. Now let’s modify the ***name*** and ***properties*** of the new ***.state*** to reflect our new View and controller:  
    })  
    ***.state('auth', {  
    url: '/auth',  
    templateUrl: 'app/auth/auth.html',  
    controller: 'AuthController',  
    controllerAs: 'auth'  
    });***
6. We will also have to ***register*** our new ***module*** inside ***index.module.js***, so let’s open that up. Copy the line that does the ***import*** of ***MainControlle***r down below itself, then let’s modify it for our new module as follows, which gives us a ***reference*** to it:  
   import { MainController } from './main/main.controller';  
   ***import { AuthController } from './auth/auth.controller';***
7. Now let’s ***register*** the Controller with ***AngularJS***. Go down into the ***angular.module()*** constructor and copy the ***.controller()*** method to below itself. Modify the new one for the new Controller:  
    .controller('MainController', MainController)  
    ***.controller('AuthController', AuthController)***Let’s give this a test in the browser by just adding ***auth*** to the end of the URL. We should get a display of our placeholder HTML.
8. We cannot see our ***Navbar*** in this view, because the application generator located it inside ***main.html*** only. Let’s cut it out of there, and load it into our ***index.html***, below the ***<body>*** tag, to make it available in all Views. Also remove the creation-date attribute, as it is no longer being used:  
    <body>  
    ***<div>  
    <acme-navbar></acme-navbar>  
    </div>***Let’s give this a test and both the ***Home*** view and the ***auth*** view should have the Navbar.
9. Let’s automate our route URL in ***navbar.html*** by modifying the ***<a> href*** for Login as follows:  
    <li><a ***ng-href="#/auth"***>Login</a></li>  
   Give that a test by clicking Login in the Navbar.
10. Let’s learn another way to do this that is provided by the ***UI Router*** we are using, which has its own special attribute, and takes a state name:  
     <li><a ***ui-sref="auth"***>Login</a></li>  
    Give that a test by clicking Login in the Navbar.
11. Now that we have set the stage, let’s start to build out our View in ***auth.html***. Let’s copy all of the code out of ***main.html*** and paste it over our placeholder in ***auth.html***. We can also modify it to remove some elements, ending up with the following, basically a container with just a panel:  
    ***<div class="container">  
     <div class="panel panel-default">  
     <div class="panel-heading">Post A Message</div>  
     <div class="panel-body">  
     </div>  
     </div>  
    </div>***
12. Now let’s wrap the panel into another Bootstrap type ***<div>*** to give a different look to the display, then copy it below itself to create two side-by-side panels, renaming one to ***Login*** and the other to Register:  
    <div class="container">  
     ***<div class="col-md-6">  
     <div class="panel panel-default">  
     <div class="panel-heading">Login</div>  
     <div class="panel-body">  
     </div>  
     </div>  
     </div>  
     <div class="col-md-6">  
     <div class="panel panel-default">  
     <div class="panel-heading">Register</div>  
     <div class="panel-body">  
     </div>  
     </div>  
     </div>***<</div>  
    Let’s regenerate the browser and take a look. This will also work on ***mobile*** devices. In the Developer Tools, click the Device Mode icon just to the left of the Elements tab.
13. Now let’s add a ***<form>*** to the ***Login*** panel as follows:  
     <div class="panel-heading">Login</div>  
     <div class="panel-body">  
     ***<form>  
     <div class="form-group">  
     <label>Email address</label>  
     <input type="email" class="form-control">  
     </div>  
     </form>*** </div>  
     </div>  
    Let’s check that in the browser.
14. Let’s copy the ***form-group*** below itself and modify it to use as a ***password*** field:  
     <div class="form-group">  
     <label>Email address</label>  
     <input type="email" class="form-control"  
     </div>  
     ***<div class="form-group">  
     <label>Password</label>  
     <input type="password" class="form-control">  
     </div>***Let’s check that in the browser.
15. We will add a ***<button>*** to the ***<form>*** to use for ***submit***:  
     ***<button type="submit"   
     class="btn btn-default">Submit</button>*** </form>Let’s check that in the browser.
16. Now let’s copy the entire ***<form>*** section down into the same position in the ***Register*** area. We will also add a copy of the password ***form-group*** down to below itself and change the ***<label>*** content on it to **Password Confirm**:  
     <div class="panel-heading">Register</div>  
     <div class="panel-body">  
     ***<form>  
     <div class="form-group">  
     <label>Email address</label>  
     <input type="email" class="form-control">  
     </div>  
     <div class="form-group">  
     <label>Password</label>  
     <input type="password" class="form-control">  
     </div>  
     <div class="form-group">  
     <label>Password Confirm</label>  
     <input type="password" class="form-control">  
     </div>  
     <button type="submit"   
     class="btn btn-default">Submit</button>  
     </form>*** </div>  
    Let’s check that out in the browser.

**Exercise06\_03\_01 – Step 2**

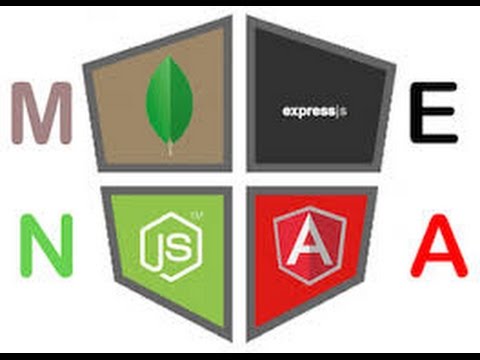
1. We will use this Exercise step to construct a ***custom AngularJS directive***. It’s purpose will be to perform a password match on User Registration to make sure that the hidden passwords are identical. We will begin by creating a new folder in the ***/app*** folder named ***directives***. Inside the folder create a new file called ***compareTo.directive.js***.
2. We will first scaffold a function to ***export*** named ***CompareToDirective()***. It will need to accept the ***$parse*** service as a parameter, and we will need to make sure it gets injected properly with an ***ngInject*** directive. The ***$parse*** service is used to convert an AngularJS ***expression*** into a ***function***:  
   ***export function CompareToDirective($parse) {  
    'ngInject'   
   }***
3. We will also have to ***register*** our new ***module*** inside ***index.module.js***, so let’s open that up. Copy the line that does the ***import*** of ***AuthControlle***r down below itself, then let’s modify it for our new module as follows, which gives us a ***reference*** to it:  
   import { AuthController } from './auth/auth.controller';  
   ***import { CompareToDirective } from   
    './directives/compareTo.directive';***
4. Now let’s ***register*** the Controller with ***AngularJS***. Go down into the ***angular.module()*** constructor and copy the last ***.directive()*** method to below itself, being sure to remove the ***semicolon***. Modify the new one for the new directive:  
    .directive('acmeMalarkey', MalarkeyDirective)  
    ***.directive('compareTo', CompareToDirective);***
5. The first step will be to go to ***auth.html*** and name our Register ***<form>***:  
    ***<form name="register">***
6. Now let’s use our new directive in the ***auth.html*** file. First we build a model for our first Password field:  
    <label>Password</label>  
    <input type="password" class="form-control"  
    ***ng-model="pwd">***
7. Next we build the model for our **Password Confirm** field and also name it:  
    <label>Password Confirm</label>  
    <input type="password"   
    class="form-control***" name=”pwd”*** ***ng-model="pwdConfirm">***
8. Now we can add our custom ***compare-to*** directive to the ***Password Confirm*** field:  
    <label>Password Confirm</label>  
    <input type="password"   
    class="form-control" ***name=”pwdConfirm”*** ***compare-to="pwd"*** ng-model="pwdConfirm">  
   Let’s check the gulp terminal, and regenerate the browser and make sure we have no errors showing up.
9. Now let’s go back to our ***compareTo.directive.js*** module and start to build out the function. To do this we will ***return*** an ***object*** that ***defines*** our directive. The first property will be ***require***, which will be set to give us access to the ***ngModel*** directive. Then we will make a property called ***link***, which will hold a ***function*** that will give us access to several ***objects*** that we will need, and the ***ngModel*** we requested:  
    'ngInject'   
    ***return {  
    require: 'ngModel',  
    link: function(scope, elm, attrs, ngModel) {  
      
    }  
    }***
10. Now we will create two ***variables***. Their purpose will be to give us ***references*** to two ***ng-models*** which we will create in our ***HTML*** for the password fields. If we analyze the code, we can deduce that ***attrs*** is holding a couple of AngularJS ***directives*** that will be in the HTML password elements, and we are going to use ***$parse*** to turn them into ***functions***:  
     link: function (scope, elm, attrs, ngModel) {  
     ***var mainModel = $parse(attrs.compareTo);  
     var secondModel = $parse(attrs.ngModel);*** }
11. Now we will build two functions, one that will be called if there is a change to the Password model, the other if there is a change to the Password Confirm model. We will use a ***$watch*** method of ***scope***, which watches for ***model changes*** and gives us the ***new*** ***value***. It will then setValidity() based upon whether the new value matches the other model or not. Let’s build the first function:  
     var secondModel = $parse(attrs.ngModel);  
       
     ***scope.$watch(attrs.ngModel, function(newValue) {  
     ngModel.$setValidity(attrs.name,   
     newValue === mainModel(scope));  
     });***
12. Let’s copy it below itself and build its ***counterpart*** to do the comparison with the other password field:  
     scope.$watch(attrs.ngModel, function(newValue) {  
     ngModel.$setValidity(attrs.name,   
     newValue === mainModel(scope));  
     });  
     ***scope.$watch(attrs.compareTo, function(newValue) {  
     ngModel.$setValidity(attrs.name,   
     newValue === secondModel(scope));  
     });***Regenerate the browser and check for errors. Now enter two different passwords for the ***Register*** form. It is still not ready, but we have no errors.
13. Now let’s build a ***<span>*** element for an ***error*** message if the passwords do not match, using an ***ng-show*** directive to control it:  
     ***<span ng-show="register.pwdConfirm.$invalid">  
     Passwords do not match.  
     </span>*** <button type="submit"   
     class="btn btn-default">Submit</button>  
    Let’s give this a browser test with different passwords, then with identical passwords. It works, but unfortunately the error message appears as soon as we put something into the first password, which is annoying.
14. To fix the issue, we can use some of our AngularJS ***validation*** ***properties***:  
     <span ng-show="***register.pwdConfirm.$dirty &&***   
     register.pwdConfirm.$invalid">  
    Let’s test that and we are good to go.

**Exercise06\_03\_01 – Step 3**

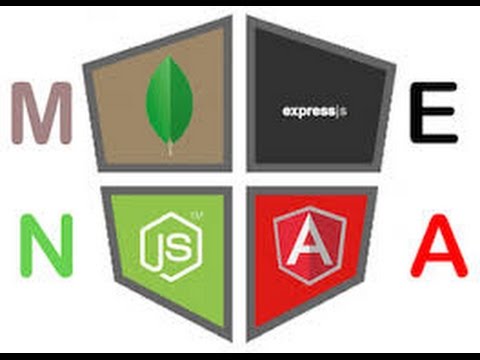
1. To build a working Email and Password Login, we will use Satellizer. Let’s shut down the ***gulp server*** to do an install. We will use the Bower package manager to take care of this. From the front-end folder, enter the following command:  
   ***bower install satellizer --save***  
   Open the IDE to the front-end and check the ***bower.json*** file. If install was successful there should be a satellizer dependency with a version number. Re-run the gulp server.
2. To load Satellizer with AngularJS, we will need to get it as a module ***dependency*** in our ***index.module.js*** file:  
   angular.module('myMessageFront', ['ui.router', 'ui.bootstrap', 'toastr'***, 'satellizer'***])
3. Satellizer is going to need to know the intended locations of our ***authentication*** ***endpoints***. We can take care of that in ***index.config.js***. First, we need to inject the Satellizer $authProvider:  
   export function config ($logProvider, toastrConfig***, $authProvider***) {  
    'ngInject';
4. We now must attach a ***property*** to the ***$authProvider*** that has the location of what will be the register endpoint, which we have not built yet. Satellizer calls this property the ***signupURL***, which we will set to empty for the minute:  
    toastrConfig.progressBar = true;  
    ***$authProvider.signupUrl = '';***
5. Let’s set up the location the clean way, by making it an AngularJS ***constant***. We can accomplish that in ***index.module.js***. Let’s set it as the first ***.constant()*** method. We will call it ***API\_URL*** and set its ***location***:  
    ***.constant('API\_URL', 'http://localhost:8080/')***  
    .constant('malarkey', malarkey)
6. Now let’s go back to ***index.config.js*** and use our new constant. To do that, we will first have to ***inject*** it:  
   export function config($logProvider, toastrConfig, $authProvider***, API\_URL***) {  
    'ngInject';
7. Now let’s use it and specify what our ***endpoint*** will be:  
    $authProvider.signupUrl = ***API\_URL + 'auth/register'***;
8. Let’s go back to ***auth.html*** and we will bind our ***register <form> submit*** to Satellizer. We can do this with an ***ng-submit*** directive in our ***<form>*** element:  
    <form name="register"  
    ***ng-submit="auth.register()"***>
9. We will have to add our ***register()*** function to our ***AuthController***, so open **auth.controller.js** and let’s build it for debug. We will do this with a ***console.log()***:  
   export class AuthController {  
    ***register() {  
    console.log('test');  
    }***}  
   Regenerate the browser with Developer tolls on Console, and give ***Submit*** a try. We should get our debug message in the Console.
10. Now let’s have Satellizer try to send a register request. We will have to build a constructor in our AuthController class to inject the $auth service:  
    export class AuthController {  
     ***constructor($auth) {  
     'ngInject';  
     this.$auth = $auth;  
     }***
11. Let’s use the ***$auth*** service ***signup()*** method in our ***register()*** function with a test object that is a fake email:  
     register() {  
     ***this.$auth.signup({email: 'test@test.com'});*** }  
    Let’s regen the browser and put the Developer Tools into the Network tab. We can see that a request goes out. If we click on it, we can see that we get a 404 Not found. That is okay, because we have not created the endpoint on our back-end yet. If we scroll down and look at the Payload, it is our test object and fake email.

**Exercise06\_03\_01 – Step 4**

1. Now that the front-end is passing Registration requests, we will need an endpoint on our server to respond to the requests. Go to the ***back-end*** and open ***server.js***. We will use ExpressJS to construct the endpoint as a ***POST***, right below our previous ***app.post()***. We will use the ***/auth/register*** URL that we set up on the front-end for Satellizer, and start with some ***debug*** to ***console.log()*** the request ***body***:  
   ***app.post('/auth/register', (req, res) => {  
    console.log(req.body);  
   });***  
   Run the server and the front-end, turn on Developer Tools in the Network tab, and try a Registration Submit. The request should show with a Status 200 and the correct Payload.
2. Now let’s work on passing the actual Email and Password from the ***<form>***. So switch back to the ***front-end*** and open up ***auth.html***. We will need to add an ***ng-model*** to the Email ***<input>*** field. Let’s make it an ***object*** so it can hold both the Email and Password, as follows:  
    <label>Email address</label>  
    <input type="email" class="form-control"  
    ***ng-model="auth.user.email"****>*
3. Let’s modify the ***ng-model*** of the Password ***<input>*** to conform:  
    <label>Password</label>  
    <input type="password"   
    class="form-control" name="pwd"  
    ***ng-model="auth.user.pwd"***>
4. Let’s modify the ***compare-to*** directive of the Password Confirm ***<input>*** to conform:  
    <label>Password Confirm</label>  
    <input type="password"   
    class="form-control" name="pwdConfirm"  
    ***compare-to="auth.user.pwd"*** ng-model="pwdConfirm">
5. We will need to modify our ***auth.controller.js*** to send the actual data. The ***signup()*** method can now send the new object:  
    register() {  
    ***this.$auth.signup(this.user);*** }  
   Give this a browser test and examine the Payload.
6. Even though we get an error message when the passwords do not match, the Submit is still enabled. We can fix this in ***auth.html*** by using an ***ng-disabled*** directive on the ***<button>***:  
    <button type="submit"   
    class="btn btn-default"  
    ***ng-disabled="register.pwdConfirm.$invalid">***  
    Submit</button>  
   Give this a browser test and we should not be able to submit with invalid passwords.

**Exercise06\_03\_01 – Step 5**

1. We would like to save our Users in MongoDB. We will have to create a ***model*** in Mongoose for them. But let’s first modularize the back-end a little bit by first creating a folder named ***Models***. Create a new file in the folder called ***Message.js***. first we will cut the ***Message*** model from ***server.js***, and paste it into the new file. Convert it from a ***var*** to a ***module.exports*** assignment. Copy the ***require()*** of Mongoose, and paste that to the top of the new file:  
   ***var mongoose = require('mongoose');  
     
   module.exports = mongoose.model('Message', {  
    msg: String  
   });***
2. We will need to ***require()*** the new module into ***server.js***, as it is used:  
   var url = 'mongodb://localhost:27017/' + dbName;  
   ***var Message = require('./models/message');***Run a browser test and everything should still be working as it was.
3. Now lets create a Model for a User. In ***/models***, create a new file called ***User.js***, and copy in the contents of ***Message.js***. Modify the code to reflect a User. Usually, we would encrypt the password for a production version, but let’s keep it simple:  
   ***module.exports = mongoose.model('User', {  
    email: String,  
    pwd: String  
   });***
4. We will need to ***require()*** the new module into ***server.js***:  
   var Message = require('./models/message');  
   ***var User = require('./models/user');***Run a server and browser test and everything should still be working as it was.
5. Let’s modify the new ***route*** to save the Registration request. We will build a User object and save it, and this time will handle any errors we receive:  
   app.post('/auth/register', (req, res) => {  
    console.log(req.body);  
    ***var user = new User(req.body);  
    user.save((err, result) =>{  
    if (err) {  
    return res.status(500).send({  
    message: err.message  
    });  
    }  
    else {  
    res.status(200);  
    }  
    });***});  
   Run the Node server and let’s test a Submit with a fake email. We should get no errors in the browser Console, and both the Node console and the browser ***Elements*** should show a good payload. Run ***RoboMongo*** and refresh. There should be a ***users*** collection and there should be good data in it.
6. The ***server.js*** file is growing again, so let’s modularize some more. Create a folder named ***controllers***, with a new file in it named ***auth.js***. Let’s scaffold out a ***register()*** function to export:  
   ***module.exports = {  
    register: () => {  
      
    }  
   }***
7. Next let’s cut out the ***/auth/register*** route callback and paste it into the auth.js ***register*** property as its function:  
    register: ***(req, res) => {  
    console.log(req.body);  
    var user = new User(req.body);  
    user.save((err, result) => {  
    if (err) {  
    res.status(500).send({  
    message: err.message  
    });  
    } else {  
    res.status(200);  
    }  
    });  
    }***
8. Now we can set it as the ***callback*** function for the route in ***server.js***:  
   ***app.post('/auth/register', auth.register);***
9. We will also need to ***require()*** our new module into ***server.js***:  
   var User = require('./models/user');  
   ***var auth = require('./controllers/auth');***
10. We will need to cut out our User ***require()*** from ***server.js*** and move it into ***auth.js*** where it is actually used, adding another ***period*** ( ***.*** ) to the path because of the new relative location:  
    ***var User = require('../models/user');***  
    module.exports = {  
    Give this a server and browser test for syntax problems.
11. Let’s some more with our **/api/message** routes where we have code for both an ***app.get()*** and an ***app.post()*** route. In ***/controllers*** let’s create a new file named ***message.js*** and scaffold out the functions to ***export***:  
    ***module.exports = {  
     get:  
     post:  
    }***
12. Let’s cut out the ***GetMessages()*** function from ***server.js*** and paste it into ***message.js*** as the value of the ***get*** property. We will also convert them to ***arrow*** function syntax. We will also cut out the ***callback*** to the ***app.post()*** route and paste it into ***message.js*** as the ***post*** property value. We also need to cut out the Message ***require()*** from ***server.js*** and paste it into ***message.js***, adding another ***period*** ( ***.*** ) to the path because of the new relative location:  
    ***var Message = require('../models/message');  
      
    module.exports = {  
     get: (req, res) => {  
     Message.find({}).exec((err, result) => {  
     res.send(result);  
     })  
     },  
     post: (req, res) => {  
     console.log(req.body);  
     var message = new Message(req.body);  
     message.save();  
     res.status(200);  
     }  
    }***
13. Now we go back to server.js and require() in our message controller:  
    var auth = require('./controllers/auth');  
    ***var message = require('./controllers/message');***
14. We need to fix up the two routes in ***server.js*** to reflect the changes, as follows:  
    ***app.get('/api/message', message.get);  
      
    app.post('/api/message', message.post);***Run the server and browser to check for errors.
15. Now let’s fix the problem of being able to Register ***duplicate*** User records in ***auth.js***. Right after the ***console.log()*** debug, let’s ***query*** the database to see if the User exists, and wrap the rest of the code in its ***callback***:  
    module.exports = {  
     register: (req, res) => {  
     console.log(req.body); ***User.findOne({  
     email: req.body.email  
     }, (err, existingUser) => {  
     var user = new User(req.body);  
     user.save((err, result) => {  
     if (err) {  
     res.status(500).send({  
     message: err.message  
     });  
     }  
     res.status(200);  
     });  
     });*** }  
    }  
    Run the server for a syntax check.
16. Now let’s trap the ***existingUser*** error:  
     }, (err, existingUser) => {  
     ***if (existingUser) {  
     return res.status(409).send({  
     message: 'Email is already registered'  
     });  
     }***Go into RoboMongo and dump the ***User*** collection. Restart the Node server and regenerate the browser. If we try to submit the same user twice, the browser Console should show our 409 error on the second one, and RoboMongo should not show only one new data record. Try to submit a different User just for a check.
17. We would like to ***authenticate*** the user on the front-end, so we will need to give the user back something for their browser to keep, with which they can identify themselves. Traditionally, we do that with ***cookies***. But the more modern technique is to use ***token authentication***. That is why we chose Satellizer. But we will need the help of a token library, specifically a ***JSON Web Token*** library, or ***JWT*** for short. We can install one on the ***back-end*** with npm, either from your IDE or a terminal in the back-end folder:  
    ***npm install jwt-simple –save***Check the package.json for results.
18. Now let’s ***require()*** it into our auth.js controller:  
    var User = require('../models/user');  
    ***var jwt = require('jwt-simple');***
19. Now let’s create a new function in ***auth.js*** named ***createToken()***, below our module.exports because it will be local to the module.:  
    ***function createToken(user) {  
       
    }***
20. Now we need to create a ***payload*** object, it will take the user **\_id**, which has been created by MongoDB. It will also need both the time that we issue the token, and when we want it to expire. To do this we will get the help of another external library called moment, which we install on the ***back-end*** with npm,:  
    ***npm install moment –save***Check the package.json for results.
21. Let’s ***require()*** in the moment library:  
    var jwt = require('jwt-simple');  
    ***var moment = require('moment');***
22. Now let’s create our ***payload*** object, using the user that is passed in to get the ***\_id***, and moment for the creation and expiration ***times***, which we will ask for in ***UNIX*** format:  
      
    function createToken(user) { ***var payload = {  
     sub: user.\_id,  
     iat: moment().unix(),  
     exp: moment().add(14, 'days').unix()  
     }***}
23. Now we can return it as an ***encoded*** token, using JWT to do the job. We will encode it against a ***token secret***, which usually would be complex and hidden in a configuration file, but we will just make it the string ***”secret”*** to keep things simple.  
     exp: moment().add(14, 'days').unix()  
     }  
     ***return jwt.encode(payload, 'secret');***
24. Now we will send back our new token with the ***Status*** ***200*** response:  
     }  
     res.status(200)***.send({  
     token: createToken(result)  
     })***;  
    Run the server and regenerate the browser, with Developer Tools in the Network tab. Register a brand new user, click on the register item, and look at the Response tab. There is our incredibly unbreakable token.
25. Now we need to get Satellizer to save that token in our browser ***local storage***. Switch to the ***front end*** and we will do the job in ***auth.controller.js***. In here, we can chain a ***Promise*** to ***signup()***, which is going out to the server via it’s ***signupUrl***, set in ***index.config.js***. That is returning our token and we can use a Satellizer method to put it in browser local storage.   
     register() {  
     ***var vm = this;  
     this.$auth.signup(this.user)  
     .then(function(token) {  
     vm.$auth.setToken(token);  
     });*** }  
    Regenerate the browser and Register a brand new user. Click on the Developer Tools Application (Resources) tab, and look at Local Storage for the URL. There is our Satellizer token.

**Exercise06\_03\_01 – Step 6**

1. Our task in this section will be to ***associate*** a user with a message post. We will need the user ID from the user who posted and is being ***authenticated*** with Satellizer. It will attach the user ***token*** in the ***authorization*** ***header*** of every ***request*** sent to the server. We can grab the user ID from that token. To do this, we will build some custom ***middleware*** in ***server.js*** called ***checkAuthenticated()***, directly below our CORES middleware. Scaffold that out as follows:  
   ***function checkAuthenticated(req, res, next) {  
      
   }***
2. First we need to check for an ***authorization*** ***header***, so let’s set the ***error*** trap as follows to look for the authorization header:  
   function checkAuthenticated(req, res, next) {  
    ***if (!req.header('Authorization')) {  
    return res.status(401).send({  
    message: 'Please make sure your request has an authorization   
    header'  
    });  
    }***  
   }
3. To ***decode*** the token, we will need to ***require()*** in our ***JWT*** library:  
   var message = require('./controllers/message');  
   ***var jwt = require(('jwt-simple');)***
4. Now let’s ***mount*** the middleware ***directly*** into our message post function as follows:  
   app.post('/api/message', ***checkAuthenticated,*** auth.register);
5. We can ***split*** off the portion of the token we need from the ***request*** ***header*** value, which is the encoded value after the word ***Bearer***. Let’s do that with the following code:   
    if (!req.header('Authorization')) {  
    return res.status(401).send({  
    message: 'Please make sure your request has an authorization   
    header'  
    });  
    }  
    ***var token = req.header('Authorization').split(' ')[1];  
    console.log(token);  
    next();***}  
   Let’s examine what this code does. Go into ***RoboMongo*** and dump the ***Users*** collection. Run the server and post a new user. In the Developer Tools ***Network*** tab, click on the second ***register*** request. Examine the ***Authorization*** header, and you can see that its value starts with ***Bearer***, followed by the Satellizer ***token***. Now go and post a new ***message***. Look in the Node console, and you can see in your debug that the ***split()*** has isolated the token.
6. Now let’s ***decode*** it, using the same ***secret*** we used to encode it. We would do this much more securely in production software. Modify the code as follows, with some more debug:  
    var token = req.header('Authorization').split(' ')[1];  
    console.log(token);  
    ***var payload = jwt.decode(token, 'secret');  
    console.log(payload);*** next();  
   Run the server and post another new message. Now look in the Node console, and you can see the ***payload*** has three decoded parts, including the ***user*** ***ID***, and ***timestamps*** of the initial token set and its expiration.
7. In order to check the expiration of the token, we are going to need to ***require()*** the ***moment*** module into the server:  
   var jwt = require('jwt-simple');  
   ***var moment = require('moment');***
8. Now let’s trap if the token has ***expired*** by checking the expiration against the current time:  
    var payload = jwt.decode(token, 'secret');  
    ***if (payload.exp <= moment().unix()) {  
    return res.status(401).send({  
    message: 'Token has expired'  
    });*** }  
    next();
9. Now let’s get to the point of this, getting the user ID into the endpoint. To do this, we can now attach it to the ***req*** object. Let’s also remove our ***console.log()*** debugs:  
    var token = req.header('Authorization').split(' ')[1];  
    var payload = jwt.decode(token, 'secret');  
    if (payload.exp <= moment().unix()) {  
    return res.status(401).send({  
    message: 'Token has expired'  
    });  
    }  
    ***req.user = payload.sub;*** next();  
   In the Developer tools, confirm that we have a Satellizer token in the Application tab. Now let’s run our server and post a new message. Look at the ***Node*** console and we should see that it went through. This can be confirmed in ***RoboMongo*** as well. Now clear the Satellizer token from Local Storage. Post a new message and the ***Console*** tab shows us a ***401*** error. If we go to the ***Network*** tab and click the red ***message***, the ***Response*** area shows us that we are missing an ***Authorization*** ***header***. Enter another new user, we get a Satellizer token, and we can again post messages. Let’s dump our entire database to start clean.
10. Let’s further ***modularize*** the app by creating a folder called ***/services***, and a file in it called ***checkAuthenticated.js***. Cut the ***jwt*** and ***moment*** ***require()*** statements out of ***server.js***, as well as the ***checkAuthenticated()*** function, and paste them into the new file. We will also ***export*** the function and clean up the ***console.log()*** debugs:  
    ***var jwt = require('jwt-simple');  
    var moment = require('moment');  
      
    module.exports = function checkAuthenticated(req, res, next) {***
11. We need to ***require()*** our new module into ***server.js***:  
    var message = require('./controllers/message');  
    ***var checkAuthenticated = require('./services/checkAuthenticated');***
12. Let’s do the same for our ***cors*** middleware into a new file in ***/services*** called ***cors.js***, and ***export*** it:  
    ***module.exports =*** (req, res, next) => {  
     res.header('Access-Control-Allow-Origin', '\*');  
     res.header('Access-Control-Allow-Headers', 'Content-Type, Authorization');  
     next();  
    }
13. We need to ***require()*** our new module into ***server.js***:  
    var checkAuthenticated = require('./services/checkAuthenticated');  
    ***var cors = require('./services/cors');***
14. Now let’s modify our ***app.use()*** to use the new module:  
    app.use(bodyParser.json());  
      
    ***app.use(cors);***Let’s give this a test in the browser and server, by adding a new user and a new message
15. Let’s run a test to make sure that we have our user ID in our ***message.js*** ***controller***. Modify it’s ***console.log()*** debug as follows:  
     post: (req, res) => {  
     console.log(req.body***, req.user***);  
    Rerun the server and add another message. The Node console should now show the user ID next to the message body.
16. We can now associate the user ID with the message post. To do this we modify our ***Message.js*** model. We will add a new ***property*** for the user ID, formatted as a Mongoose-style data type ***ObjectId***. The ***ref:*** property with the ***User*** string gives Mongoose a reference to the ***User*** model we built as part of the Mongoose ***schema***:   
    module.exports = mongoose.model('Message', {  
     msg: String,  
     ***user: {type: mongoose.Schema.ObjectId, ref: 'User'}***});
17. Now we can modify the ***message.js*** controller to add the user to the ***req.body*** so it will get saved by Mongoose:  
     post: (req, res) => {  
     console.log(req.body, req.user);  
     ***req.body.user = req.user;***Let test by rerunning the server and posting a new message. Go into RoboMongo and check the new message which should now have a user ID.
18. Now we can use a feature of Mongoose to ***populate*** our ***Message.find()*** call with data from the ***User*** object associated with it. This acts in a similar fashion to a relational database ***join***. Notice that it has syntax by which we can tell it what fields not to use on the populate, in this case ***pwd***:  
     get: (req, res) => {  
     Message.find({})***.populate('user', '-pwd')***.exec((err, result) => {  
     res.send(result);  
     })  
     },  
    Run the server and refresh the Home screen to display the messages. Go to the Developer Tools Network tab, click on ***message*** and ***Response*** to view the response. Scroll all the way to the last message and we will see that it has a user attached to it. the user should now have the user ID and the email, but not the password.
19. We would like to display the email with the user messages. Leave the server running and switch to the ***front-end***. We can do some work on ***main.html***. We will add an AngularJS expression to display the email:  
     <li class="list-group-item"  
     ng-repeat="message in main.messages">  
     {{message.msg}} ***{{message.user.email}}***  
     </li>  
    Regenerate the browser and the email should show up.